Efficient Bill-Of-Materials Algorithms

Ahmad Khalaila[*], Frank Eliassen
Department of Computer Science
The University of Tromsø
9037 Breivika, Norway
{ahmad,frank}@cs.uit.no

Catriel Beeri
Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
beeri@cs.huji.ac.il

September 1, 1996

Abstract

It has been shown that every linearly recursive database query can be expressed as a transitive closure, possibly preceded and followed by relational algebraic operations. A large class of such queries computes the bill-of-materials of database relations.

This paper presents efficient sequential and distributed algorithms that compute the bill-of-materials of a database relation. These algorithms use a special join operation that accumulates the cost of composite parts, without constructing the transitive closure of the argument relation, thus saving time and space.

Moreover, the distributed algorithm is very efficient in terms of communication complexity. The number of tuples exchanged between the sites is neither dependent on the size of the argument relation nor on the size of its transitive closure of that relation. That number is simply equal the number of different parts in the argument relation. For the distributed setting we develop both a synchronous and an asynchronous algorithms and their analysis. We conclude that partially synchronous algorithms seem to be superior to both of them.

1 Introduction

Given a transitive closure operator denoted $BAG - TC$ that does not eliminate redundant paths, and the relations defined by the following relational schemes:

$$Uses = (part : oid, subpart : oid);$$

$$Base = (part : oid, cost : real);$$

where $Uses$ is transitively defined and has a tuple for each ($part, subpart$) relationship. A composite part may be involved in many such tuples. The $Base$ relation has a tuple for each base part (i.e. a part which is not composed of any other parts).

[*]part of this work was done while Khalaila A. was financed by The Royal Norwegian Research Council

To compute the Bill-Of-Materials (BOM) we evaluate the expression:

$$\prod_{Uses.Part,sum(Cost)} (GroupBy_{Uses.Part,sum(Cost)}(Base \bowtie_{Base.Part=Subpart} (BAG-TC(Uses))))$$

An execution strategy for the above expression that is based on evaluating each operation in the (above) strict nested order incurs very high execution cost. This high cost is due to the intermediate construction of the transitive closure of $Uses$. We argue that any execution strategy for BOM algorithms that constructs the transitive closure of its argument is a bad strategy, in particular when that closure is much larger than the given relation.

In this paper, we present BOM algorithms that avoid the evaluation of the transitive closure operator, and combine some of the operations mentioned above into one specialized join operation, called CJOIN.

Based on this combined join algorithm we develop three BOM algorithms; an iterative one, a synchronous distributed one, and an asynchronous distributed algorithm. We also (informally) prove their correctness, and analyze their complexity

Some improvements to both the synchronous and the asynchronous algorithms are proposed, as a result of their complexity analysis. We conclude that a partially synchronous algorithm will in general outperform both the synchronous and the asynchronous algorithms.

The distributed algorithms assume the availability of a network broadcast capability. However, an idea is presented at the end on how to make these algorithms independent of such a capability.

1.1 Related Work

It is a widely known fact that query languages based on relational algebra or calculus [Codd70, Codd72] are limited in their expressive power especially in expressing recursive queries [Ullm88a].

A transitive closure operator for database queries was first proposed by Zloof in [Zloo75]. Since then it has been shown that linear recursive queries can be expressed by such an operator [JaAN87, ChHa82], and an extension of relational algebra that includes a transitive closure operator called α-algebra has been proposed in [Agra87]. Furthermore, Agrawal (as well as many others) proposed that specialized algorithms that exploit the knowledge of the physical database can be built into the database system to efficiently implement the transitive closure operator and some frequent applications of it.

Bill-of-materials computations constitute a large class of linearly recursive database computations, that occur frequently in database systems environments containing transitive relations. When such queries are applied to very large relations, their efficient processing become vital (e.g. for users that are highly dependent on them). Although all such computations can be expressed using the transitive closure operator (as has been illustrated above), evaluating the transitive closure is not necessary for the evaluation of such computations. Since such an evaluation often incurs a very high cost in terms of time and space, we would like to avoid it. This is very similar to avoiding the evaluation of the cross-product when join is being evaluated [SmCh75, Ullm88b].

Many efficient algorithms have been developed for different computing environments [Tarj81, AgJa87, Lu87, BiSt88, IaRa88, AgJa88, VaKh88a, VaKh88b, AgDJ90, ChDe90, HoAC90, Jian90, Jako91, DaJa92]. In this paper we propose closure-based (BOM) algorithms that avoid constructing the transitive closure of the argument relation. This results in better performance, both in terms of time and space.

Traditionally, the optimizer in a query processing system starts by algebraic manipulations that optimize the query tree. Then methods implementing the various operations in the query tree are chosen using cost estimation and the current access paths existing in the physical database [SmCh75, SACLP79]. Combining the execution of many operations into one has been first proposed by Smith et al. [SmCh75], and since then has been adopted by nearly everyone working with query processing and optimization [JaKo84, Ullm88b]. The combined join (CJOIN) algorithm is the core of our BOM algorithms since it combines the accumulation of (intermediate partial) cost for composite parts using the cost of their subparts, with the binary matching normally applied in join operations, to avoid the intermediate construction of the transitive closure of the input relation.

1.2 Organization

The presentation is organized in two main parts. The first one, which is presented in section 2, is concerned with the implementation, correctness and complexity analysis of the CJOIN and the iterative algorithm. The second part, which is presented in section 3, is concerned with the data partition, implementation, correctness, and complexity analysis of the distributed algorithms.

In section 4, we propose a number of improvements to the distributed algorithms, and discuss some future work, while section 5 concludes.

2 The Iterative BOM Algorithm

To compute the bill-of-materials for all the composite parts present in the $Uses$ relation, it is not necessary to perform the transitive closure operation present in the BOM expression above, since we are not interesting in the all-pairs transitive closure of the graph represented by the $Uses$ relation.

Additionally, many of the operations involved in the above query, can be done in a combined join operation (called CJOIN). The operation tries to match the subpart attribute of each tuple in the $Uses$ relation with the part attribute of each tuple in the $Base$ relation. If a match occurs it partially performs the *sum* operation by accumulating the cost of a $Uses$ composite part that have a subpart that match a base tuple. The cost for each part is accumulated in the cost attribute of the corresponding tuple of the temporary relation *Accum*, which states the identity and cost accumulated so far for each (composite) part. The relational schema of *Accum* is

$$Accum = (part : oid, cost : real);$$

When analyzing the composition relationship we found that some parts are not composed (i.e. they are atoms or base parts), some parts are composed only of base parts (we will call them 1^{st} level parts), some parts are composed only of base (i.e. 0-level) and 1^{st} parts (we call them 2^{nd} level parts), some parts are composed only of 0-level, 1^{st} level, and 2^{nd} level parts (we call them 3^{rd} level parts), and in general i^{th} level parts are composed only of parts from the levels below, i.e. 0-level, 1^{st}level, 2^{nd}level, $\ldots$, and $i-1$ level. Notice that the sets of parts from the different levels are disjunctive.

Based on the above observation, the iterative BOM algorithm (IBOM) starts by computing the total cost for 1^{st} level parts, then the total cost for 2^{nd} level parts, and so on. In general, computing the total cost for parts from the i^{th} level, will be completed only after the total cost for all parts from all the levels below (i.e. $i-1, i-2, \ldots, 1$) have been computed. Therefore, a run of the iterative BOM algorithm consists of the subsequent phases $1, 2, \ldots, D$, where D denotes the diameter of the directed acyclic graph (DAG) as represented by $Uses$. In each phase the total cost for the parts from the corresponding composite level are computed. That is, in phase i the total cost for all the parts from level i are computed, and phase i (for $i > 1$) is preceded by phase $i-1$ and is followed by phase $i+1$ (for $i < D$). Such a BOM algorithm terminates after the D^{th} phase.

2.1 Implementation of CJOIN

In this section we develop the CJOIN operation used in the IBOM algorithm specified below. This operation takes as input three argument relations $Accum, Uses$, and $Base$, and delivers as output three argument relations $Accum, Uses$, and $NewBase$.

The tuples of $Uses$ are grouped by the *part* attribute, and $Base$ and $Accum$ are hashed on their *part* attributes.

The following five operations are needed to implement the CJOIN operation. The signatures and informal semantics of these operations are given below:

- $match : oid, Base \rightarrow TupleOf(Base)$
  match takes a part identity as its first argument and the current $Base$ relation as it second argument, and returns the $Base$ tuple corresponding to its first argument.
- $accumulate : real, oid, Accum \rightarrow$
  the $Accum$ tuple corresponding to its second argument is looked up, and its cost attribute is incremented by the value of the first argument. If such a tuple does not exist, it is created and inserted into $Accum$ and its cost attribute is initialized to the value of the first argument.
- $mark\_del : TupleOf(Uses), Uses \rightarrow$
  this function puts a deletion mark on the $Uses$ tuple corresponding to its first argument. This operation shrinks the volume of $Uses$.
- $move\_2NewBase : TupleOf(Uses), Accum, NewBase \rightarrow$
  this operation is called when the total cost for a composed part has been computed completely. It moves the $Accum$ tuple corresponding to its first argument to $NewBase$. This is the operation that extracts the base tuples of the next phase of the IBOM algorithm, and inserts them into $NewBase$.

The above operations are implemented on top of hash-based structures on $Base$ and $Accum$. Hash-based structures and algorithms has been designed mainly to speed up the join operation involved in the IBOM algorithm[Brat84, Kits83].

The $CJOIN$ algorithm performs the join of $Uses$ and $Base$, reduces and reconstructs all its arguments relations, and partially computes the aggregate function sum, all in one run through the tuples of $Uses$, $Base$, and $Accum$.

```
**Algorithm** CJOIN
**Input:** $Uses, Base, Accum$;
**Declare:**
    $u : TupleOf(Uses)$;
    $b : TupleOf(Base)$;
```

```
        $p : TupleOf(Accum)$;
        $u2Base : Bool$;
    { This flag indicates if the total cost of $u$ has been computed, or not.}
    **Begin**
        **Foreach** $group \in GroupBY_{part}(Uses)$ **Do**
            $u2Base \leftarrow True$;
            **Foreach** $u \in group$ **Do**
                $b \leftarrow match(u.subpart, Base)$;
                **If** $b \neq \perp$ **Then**
                **Begin**
                    $accumulate(b.cost, u.part, Accum)$;
                    $mark\_del(u, Uses)$;
                **End**;
                **Else**
                    $u2Base \leftarrow False$;
            **Od**; { Foreach u }
            **If** $u2Base$ **Then**
                $move\_2NewBase(u, Accum, NewBase)$;
        **Od**; {Foreach group ...}
        $Return(Accum, Uses, NewBase)$;
    **End**.
```

2.2 Notations and assumptions

In the sequel we will use the following notations and assumptions:

- $|Uses| = N$, denotes the number of tuples of the $Uses$ relation;
- $|Uses^i| = N^i$, denotes the number of (remaining) tuples in $Uses$ at the end of the i^{th} phase;
- I is the number of distinct part identities that occur in the part attribute of $Uses$; i.e., the number of groups in $GroupBy_{part}(Uses)$;
- $|Base| = M$, denotes the number of tuples initially in $Base$;
- $|Base^i| = M^i$, denotes the number of tuples in $NewBase$ at the end of the i^{th} phase;
- The auxiliary operations $match$, and $move\_2NewBase$ have a constant cost, denoted by C_0, while the others have a neglectable cost. C_0 actually denotes the cost of accessing a tuple in $Base$ or $Accum$;
- C_1 denotes the cost of accessing a $Uses$ tuple;
- C_2 denotes the cost of broadcasting a tuple of $Base$;

A simplifying assumption that otherwise has no major implication is the following:

Assumption 2.1 *(Uniform CJOIN behavior) The complexity of CJOIN behavior at the different D phases is uniform. That is, the same number of tuples are added to new Base, and Accum and the same number of tuples are deleted from $Uses$, at each phase.*

2.3 The Cost Formula for CJOIN

The CJOIN algorithm consists of two loops, one through the different groups in $Uses$ and the other through the tuples of these groups. By assumption, there are I different groups in $Uses$ corresponding to the I different composite parts in $Uses$. Each of these groups has in average N/I tuples. Therefore the total number of tuple accesses to $Uses$ is N. The maximum number of accesses to $Base$ is N, and the number of actual *Accum* accesses is equivalent to the number of tuples deleted from $Uses$ (in the same phase), which is approximately N/D (based on our previous assumptions). The number of accesses to $NewBase$ is equivalent to the number of new $Base$ tuples which is approximately I/D. Consequently, the cost formula for CJOIN is defined as follows:

$$CF_{CJOIN} = N \times C_1 + N \times C_0 + (N/D) \times C_0 + (I/D) \times C_0$$

In the above formula, the first and second terms denote the cost of the hash-based join operation. The third term, $(N/D) \times C_0$, correspond to the cost of accessing the *Accum* tuples in order to accumulate the cost of their corresponding parts. The fourth term $(I/D) \times C_0$ corresponds to the cost of restructuring *Accum* and $NewBase$.

Notice that the number of tuples in *Accum* will never exceed the number of groups in $Uses$ which is equivalent to I, however the number of tuple accesses to *Accum* can be much higher than I. Moreover, the number of tuples in *NewBase* will never exceed I, in average it will be I/D.

Since $N > I$ is always true, the above formula is rewritten to:

$$\begin{aligned} CF_{CJOIN} &= N \times (C_1 + C_0) + (2N/D) \times C_0 \\ &= N(C_1 + C_0 + 2C_0/D) \end{aligned} \tag{1}$$

2.4 Implementation of the iterative BOM algorithm

The iterative BOM algorithm can be seen as a loop of joins between the $Base$ and the $Uses$ relations, each of which corresponds to a phase, as defined above. In each iteration the contents of the two relations will be changed, as explained in the sequel. Initially, the base parts will be those in $Base$, and $Uses$ will have all the tuples representing the (*part*, *subpart*) relation.

In the first iteration the total cost for all parts from 1^{st} level will be computed, the cost for all other parts that have some base subparts will be accumulated in *Accum*, every tuple in $Uses$ that has a base subpart will be (marked) deleted, and the 1^{st} level parts together with their total costs comprise the new $Base$ (denoted $Base^1$) of the next phase.

In the second iteration, the total cost for all parts from 2^{nd} level will be computed as above, and in general, in the i^{th} iteration the total cost of all parts from the i^{th} level will be computed, the cost of all other (i.e. higher levels) parts that have some base part components will be accumulated in *Accum*, every tuple in $Uses$ that has a $Base^{i-1}$ subpart will be (marked) deleted, and the i^{th} level parts together with their cost comprise the new $Base$ of the next phase (denoted $Base^i$).

The IBOM algorithm depicted below constructs in each iteration (i) a new logically separated relation (fragment) to contain the new base tuples, and is called $Base^i$. That is, the base fragment $Base^i$ is constructed at the i^{th} iteration and corresponds to the $Base$ relation of iteration $i + 1$. $Base^i$ contains a tuple for each of the i^{th} level part which has a part attribute corresponding to that i^{th} level part and a cost attribute whose value is the total cost of that part. $Base^0$ corresponds to the initial $Base$ relation which is used in the first iteration.

The temporary relation *Accum* will at the end of each iteration i contain the cost for each j^{th} level ($j > i$) part which have some subpart from the levels below i. Within the i^{th} iteration, when the total cost for a level i part is computed, the *Accum* tuple corresponding to that part, is moved from *Accum* to $Base^i$.

Finally, the *Uses* relation will at the end of each iteration i, have no tuple with a subpart from level i or any level below.

```
Algorithm IBOM
Input: $Accum, Uses, Base$;
Begin
    $i \leftarrow 1$;
    $result \leftarrow Base^0$
    While($Uses^i \neq \emptyset$) Do
        $(Accum^i, Uses^i, Base^i) \leftarrow CJOIN(Accum^{i-1}, Uses^{i-1}, Base^{i-1})$;
        $result \leftarrow result \cup Base^i$;
        $i \leftarrow i + 1$;
    Od
    Return($result$);
End.
```

2.5 Correctness of IBOM

To prove the correctness of the above algorithm we have to show that it computes the total cost for all parts, and then terminates. That is, the total cost for all parts from level $1, 2, \ldots, D$, and that the algorithm terminates after computing the total cost for all level D parts, because at that point *Uses* become empty.

Our argument follows the informal operational specification of the algorithm given above. That is, in a run of the algorithm there are D phases. In the first phase the total cost of all parts from the first level are computed, and these parts together with their total cost constitutes $Base^1$. In general in the i^{th} phase the total cost of all parts from the i^{th} level are computed, and these parts together with their total cost constitutes $Base^i$. Finally in the last phase D the total cost of all parts from the D^{th} level are computed, and these parts together with their total cost constitutes $Base^D$.

Notice that after each phase i the total cost of all parts from all levels up to the i^{th} have been computed. When the D^{th} phase is completed the total cost of all the parts from all levels up to the D^{th} level have been computed, and the union of the base fragments will constitute the result of our BOM computation; i.e. the result returned by the IBOM algorithm is

$$\bigcup_{i=0}^{D} Base^i$$

The above arguments (informally) prove the so-called partial correctness of the algorithm, but it does not prove total correctness. To prove the total correctness of the algorithm we have to prove its termination property, which is a liveness property. This requires that the underlying (concurrent) system that executes the algorithm is fair. Here, we just assume the underlying system to be fair, for a comprehensive treatment of fairness see [Fran86]. To prove that IBOM terminates when phase D is completed, it is sufficient to prove that $Uses$ become empty at that point. That is because it is obvious from the above specification of the algorithm that it terminates when $Uses$ is empty.

Let us now look at how $Uses$ shrinks at each phase. As mentioned early, in each phase i every $Uses$ tuple that have a $Base^{i-1}$ part as a subpart is (marked) deleted. Since a run consists of the consecutive phases $1, 2, \ldots$, and D, the tuples that are still in $Uses$ after the i^{th} phase are only the tuples from the levels above i. That is, after phase D is completed the only tuples in $Uses$ will be those from level $k > D$. Since D is the diameter of the graph represented in $Uses$ then it has no tuple from any level above D. Consequently, when IBOM completes phase D and tries to go into a new phase, it will terminate since $Uses$ become empty.

2.6 The Cost Formula of IBOM

Intuitively, the cost of the join operation of the i^{th} iteration is cheaper than that of all the preceding iterations, because that cost is strongly dependent on the number of tuples in $Uses$ (i.e. N), and N is reduced in each call to CJOIN by N/D.

The cost formula for the iterative BOM algorithm can be expressed by using the cost formula previously developed for CJOIN, as follows:

$$\begin{aligned} CF_{IBOM} &= \Sigma_{i=1}^{D} N_i(C_1 + C0 + 2C_0/D) \\ &= (C_1 + C0 + 2C_0/D)\Sigma_{i=1}^{D}(N_i) \end{aligned} \tag{2}$$

The above formula is derived simply from the fact that in a run of IBOM there is an CJOIN call (whose cost is defined by equation 1) for each of the D levels in the DAG represented by $Uses$.

The term $C_1 + C_0 + 2C_0/D$ in CF_{IBOM} involves only constants and therefore cannot be reduced further. However, exploiting assumption 2.1, we may set $N_i = N - (i-1)N/D$. The term $\Sigma_{i=1}^{D}(N_i)$ can then be reduced as follows:

$$\begin{aligned} \Sigma_{i=1}^{D}(N_i) &= N_1 + \ldots + N_D \\ &= N + (N - N/D) + (N - 2N/D) + \ldots + N - (D-1)N/D \\ &= ND + (-N/D - 2N/D - \ldots - (D-1)N/D) \\ &= ND - (1 + 2 + \ldots + (D-1))N/D \\ &= ND - ((D-1)(1 + (D-1))/2)N/D \\ &= ND - (D-1)N/2 \\ &= N(D - (D-1)/2) \\ &= N(D+1)/2 \end{aligned} \tag{3}$$

Finally, by substituting equation 3 into equation 2 (i.e. (N(D+1)/2) for $\Sigma_{i=1}^{D}(N_i)$) we get:

$$CF_{IBOM} = (C_1 + C0 + 2C_0/D)(D+1)N/2 \tag{4}$$

3 The Distributed BOM Algorithms

The distributed bill-of-materials algorithms (DBOM) presented here, are in concept based on the same ideas developed for the iterative algorithm. However, these algorithms are motivated by different assumptions concerning the system model and its “normal” behavior.

3.1 System Model

The system consists of a collection of n autonomous (database server) sites, denoted $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, interconnected by a communication network with a broadcast capability. For the time being we will assume both the sites and the network channels to be reliable (i.e. fault-free).

The DBOM algorithms presented here are symmetric, i.e. the same module is running at the different sites, but with different site identities, and different sets of data. Moreover, all the sites send different data but receive the same data. A DBOM algorithm consists of the modules $DBOM_1, DBOM_2, \ldots$, and $DBOM_n$, where $DBOM_i$ is the module running at site p_i, and is capable of communicating (by broadcast) with all the other modules located at the other sites in the system.

3.1.1 Data Distribution

The data distribution is based on group partitioning of $Uses$, and replication of the $Base$ tuples for each level at each iteration. This strategy has been chosen in order to enable a site to determine if the total cost for a part has been computed or not, without any communication with other sites. This is possible since the whole group of $Uses$ tuples that defines all the subparts of that part are located at the same site.

We assume the distribution of the groups in $Uses$ is based on a hash function on the *part* attribute, that distributes the tuples *uniformly* among the sites involved. We also assume the *Accum* relation being distributed according to the same hash function applied on the *part* attribute. However, this last assumption is not strictly needed. Recall that *Accum* is a temporary relation which is initially empty and will also be empty when the computation terminates. The final result will be constructed by taking the union of all the fragments of the $Base$ relation.

3.1.2 Intuition for DBOM

The data distribution described above locates disjunctive subsets of the groups in $GroupBy_{part}(Uses)$ at the different sites. Recall that each such group of tuples defines the subparts involved in producing the part that "owns" that group. Each site p_i will be responsible for computing the total cost for the "owners" parts located at that site. To do that, site p_i may have to get the total cost of some subparts of that owner part from other sites, where their groups are located. The above implies that each site will be computing the cost for a subset of parts that is completely disjunct from the subsets of parts for which the total cost is computed at the other sites.

3.1.3 Assumptions

Below, we state 3 assumptions. The main objective of putting forward the first assumption is to ease our analysis, understanding, and development of the algorithms and their complexities. However, the remaining assumptions have a major impact on the execution of the algorithms, which will be discussed later.

Assumption 3.1 *(Uniform distribution among groups) The $Uses$ tuples are (approximately) uniformly distributed among the different groups in $GroupBy(Uses)$.*

Assumption 3.2 *(Uniform distribution among sites) The $Uses$ relation is partitioned and distributed uniformly among the sites in the system, thus each site has approximately the same number of $Uses$ groups (and tuples).*

Assumption 3.3 *(Uniform diameter among sites) Groups of parts from all the levels of the DAG, are attached in a uniform manner to each site. That is, each site has the same number of part groups from each level of the DAG represented in $Uses$.*

3.1.4 Complexity

In the preceding sections, when developing the cost formula of the CJOIN and iterative BOM algorithms, the focus is on the number of tuple accesses to each relation involved in a run of the algorithm.

For the distributed setting, in addition to the number of tuple accesses involved, the communication complexity in terms of the number of tuples exchanged between the sites over the network channels, and the number of communication phases in a run, must be taken into account.

The communication complexity assumes that the communication cost among different channels in the network is uniform.

3.2 Synchronous DBOM Algorithm

Similar to the IBOM algorithm, a run of the synchronous DBOM (SDBOM) algorithm consists of D phases, in each of which the total costs for all the parts from the corresponding level are computed, i.e. in phase i the total costs for all the parts from the i^{th} level are computed.

Recall that $Base^0$ is replicated at all the sites, which implies that each site p can now carry out its first phase to compute the total cost for all the local first level parts.

Generally, in the i^{th} phase the total cost for all local i^{th} level parts are computed, the cost of all other local parts that have some $Base^{i-1}$ components will be accumulated in the local $Accum$. Every tuple in the local $Uses$ that has a $Base^{i-1}$ subpart will be (marked) deleted, and the local i^{th} level parts together with their total cost ($Base^i_p$) comprise a fragment of $Base^i$ used in the next phase by all the sites. Thus $Base^i_p$ is broadcast by $DBOM_p$. Moreover, all the $Base^i$ fragments broadcast by the other sites are received at p so that $Base^i$ contains all the second level parts in the entire system, and then the next $(i+1)$ phase can start.

3.2.1 Implementation of the synchronous DBOM Algorithm

In our implementation each $SDBOM_p$ will in the i^{th} phase perform the following actions:

$Cjoin^i_p$: a call to $CJOIN$ in phase i by site p (denoted $Cjoin^i_p$) computes the total cost for all local i^{th} level parts, accumulates the cost for all other local parts that have some $Base^{i-1}$ subparts in the local $Accum$, and every tuple in the local $Uses$ that has a $Base^{i-1}$ subpart will be (marked) deleted.

The $Cjoin^i_p$ event produces a relation (fragment) $Base^i_p$ which contains a tuple for each i^{th} level part consisting of its identity and its cost.

$Bcast_p^i$: the $Base_p^i$ produced by $Cjoin_p^i$ is a fragment of $Base^i$ which comprise the base tuples of the next phase and is going to be used by all the sites. Therefore $SDBOM_p$ broadcasts $Base_p^i$ to all sites by a call to *Broadcast* which generates the $Bcast_p^i$ event.

$RecvAll_p^i$: all $Base^i$ fragments broadcast by the other sites are received by p so that $Base^i$ contains all the i^{th} level parts in the entire system. $RecvAll_p^i$ results in the receipt of $Base_{p_1}^i, Base_{p_2}^i, \ldots$, and $Base_{p_n}^i$. It calls the action $Receive(\mathcal{P})$ for each remote site $r \in \mathcal{P}$ to receive $Base_r^i$, and it will wait until $Base_p^i$ for all remote sites p have been received. This is implemented by the **Do** ... **Receive** ... **Until** statement in the SDBOM algorithm depicted below.

This action leads to the synchronous nature of the algorithm, because all sites will wait until they receive all the *Base* fragments produced at all the other sites before they go on with the next phase.

Since the algorithm imitates behavior of the *IBOM* algorithm, a run of *SDBOM* terminates after D phases. However, some sites may not have parts from any level above some k, where $k < D$, so the BOM computation at these sites terminates after k phases.

In the rest of the paper, we will assume that all sites have parts from all levels up to D. This is necessary to avoid that runs of the algorithm get stuck waiting for some base fragments that do not exist. Notice that this could alternatively be solved by having each site p that has produced an empty $Base_p^i$ send a dummy message indicating that $Base_p^i$ is empty.

A site p that has completed its computation (i.e. when the total cost for all its parts have been computed) informs the other sites by issuing a $Broadcast_p(Terminated)$ action. Such a terminated site will be excluded from the set of active sites $\mathcal{P}$.

```
Algorithm $SDBOM_p$
Input: $Accum_p, Uses_p, Base$;
Begin
    $i \leftarrow 1$;
    $\mathcal{TP} \leftarrow \mathcal{P}$
    While($Uses_p^i \neq \emptyset$) Do
        $(Accum_p^i, Uses_p^i, Base_p^i) \leftarrow CJOIN(Accum_p^{i-1}, Uses_p^{i-1}, Base^{i-1})$;
        $Broadcast_p(Base_p^i)$;
        $Base^i \leftarrow Base^i \cup Base_p^i$;
        Do
            $(r, Base_r^i) \leftarrow Receive(\mathcal{TP})$;
            $Base^i \leftarrow Base^i \cup Base_r^i$;
            $\mathcal{TP} \leftarrow \mathcal{TP} - r$
        Until $\mathcal{TP} = \emptyset$;
        $i \leftarrow i + 1$;
        $\mathcal{TP} \leftarrow \mathcal{P}$
    Od;
    $Broadcast_p(Terminated)$;
End.
```

3.2.2 Correctness for SDBOM

In each phase i of an SDBOM execution, $CJOIN^i$ is performed at each site and a new set of $Base^i$ tuples (to be used in the next phase) is produced and broadcast to all other sites. Each site will then receive all the new set of $Base$ tuples produced at all the other sites in this phase, which union constitutes the $Base$ used in the next phase.

The phase i in a typical run of SDBOM will look like

$$Cjoin^i_{p_1}; Bcast^i_{p_1}; \dots; Cjoin^i_{p_n}; Bcast^i_{p_n}; RecvAll^i_{p_1}; \dots; RecvAll^i_{p_n};$$

If we take the computational projection of the above schedule, which will consists only of the computational events (*i.e.* communication events are discarded from a schedule to obtain its computational projection), the schedule will look like

$$Cjoin^i_{p_1}; \dots; Cjoin^i_{p_n};$$

The computational projection of SDBOM in a phase i is computationally equivalent to the $CJOIN^i$ of the iterative BOM algorithm, and they both compute the total cost for all i^{th} level parts, accumulate the cost for all other parts that have some $Base^{i-1}$ subparts in $Accum$, and every tuple in $Uses$ that has a $Base^{i-1}$ subpart will be (marked) deleted. Both will also form a relation $Base^i$ which contains a tuple for each i^{th} level part consisting of its identity and its cost. However, in SDBOM the local fragments of $Base^i$ are exchanged among the sites (by the communication events) before it is formed.

The correctness argument for the synchronous DBOM algorithm follows along the same lines as those for the iterative BOM algorithm, because both of them are based on D consecutive phases, in each of which the total cost for all the parts from the corresponding level are computed. However, the underlying system for the distributed setting is much more complicated than that of the iterative one. We have actually eliminated a great deal of complexity by assuming and designing the computational phases to be synchronous, and assuming the reliability of the sites and the network.

3.3 Complexity Analysis for SDBOM

The cost formula for the synchronous DBOM algorithm denoted CF_{SDBOM_p}, is based on the fact that the executions of the different phases and the exchange of their results are (approximately) synchronous. This implies that a run of SDBOM is a series of phases of length D (as for the iterative case). The SDBOM run fragment corresponding to phase i looks as follows:

$$Cjoin^i_{p_1}; Bcast^i_{p_1}; \dots; Cjoin^i_{p_n}; Bcast^i_{p_n}; RecvAll^i_{p_1}; \dots; RecvAll^i_{p_n};$$

Notice that the events in the above run fragment are partially ordered according to two rules; First, a $Cjoin^i_p$ event always precede a $Bcast^i_p$ event, and secondly, a $RecvAll^i_p$ is always preceded by a $Cjoin^i_p$ and $Bcast^i_p$ events for all sites in the system, in other words, a $Cjoin^i_p$ event will always precede a $RecvAll^i_q$ event, and also a $Bcast^i_p$ event will always precede a $RecvAll^i_q$ event, for any site(s) p, q, and any phase i. The $Cjoin^i_p$ and $Bcast^i_p$ events of different sites may occur simultaneously.

A complete SDBOM run looks as follows:

$$\begin{array}{c} Cjoin^1_{p_1}; Bcast^1_{p_1}; \dots; Cjoin^1_{p_n}; Bcast^1_{p_n}; RecvAll^1_{p_1}; \dots; RecvAll^1_{p_n}; \\ \vdots \\ Cjoin^D_{p_1}; Bcast^D_{p_1}; \dots; Cjoin^D_{p_n}; Bcast^D_{p_n}; RecvAll^D_{p_1}; \dots; RecvAll^D_{p_n}; \end{array}$$

The partial order rules specified above still apply, in addition to the rule stating that any $Cjoin_p^i$, $Bcast_q^i$, or $RecvAll_r^i$ event always precede any $Cjoin_p^{i+1}, Bcast_q^{i+1}$, or $RecvAll_r^{i+1}$ event, for any site p, q, or r, and any phase $i < D$. That is, any event of the i^{th} phase precedes any event of phase $i+1$.

Each row of the above run roughly represents the actions that occur at all sites in a specific phase, where the i^{th} row corresponds to the i^{th} phase. The cost formula for the SDBOM algorithm at a site p denoted by CF_{SDBOM_p}, is stated by the following equation:

$$\begin{aligned} CF_{SDBOM_p} &= \Sigma_{i=1}^{D}(C_1 + C_0 + 2C_0/D)N_i/n \\ &= (C_1 + C_0 + 2C_0/D)/n\Sigma_{i=1}^{D}N_i \\ &= ((C_1 + C_0 + 2C_0/D)/n)N(D+1)/2 \\ &= ((C_1 + C_0 + 2C_0/D)(D+1)N/2n \end{aligned} \tag{5}$$

The right hand side of the above equation specifies the cost of all the executions of CJOIN occurring at site p, which is defined in terms of number of tuple accesses.

There is no need for further analysis or reduction of the first (multiplicative) term. We just observe that the cost specified by this term is equal the cost of the iterative algorithm divided by n (i.e. the number of the sites). That is a consequence of our assumptions about uniformness and synchrony.

The total cost of SDBOM is the sum of the costs at all sites, while the response time is measured by the maximum of these costs.

The communication complexity for SDBOM denoted by CCF_{SDBOM_p} is defined as follows:

$$\begin{aligned} CCF_{SDBOM_p} &= \Sigma_{i=1}^{D}C_2(M_p^i) \\ &= C_2\Sigma_{i=1}^{D}(M_p^i) \\ &= C_2(I/n) \end{aligned} \tag{6}$$

The above formula stands for the cost of sending all the new base tuples generated at site p, to all the other sites. Since a broadcast capability is assumed, the cost of sending a message to all sites in the network is a constant C_2 independent of the number of sites.

The last reduction is based on the observation that the number of new base tuples generated at (and broadcast by) a site is equal to the number of groups resident at that site. Moreover, our assumption concerning the uniform distribution of groups to sites imply that each site has I/n groups, therefore the last reduction to $C_2(I/n)$.

In a run of the algorithm at all sites, a new base tuple is generated for each group only once. Thus, the total number of base tuples to be broadcast in a complete run of the algorithm is equal to I. This is a very interesting result, because it shows that the communication complexity of the SDBOM algorithm is dependent only on the number of groups and not on the number of tuples in $Uses$.

When analyzing the above formula, we notice that whenever the number of groups in $Uses$ is very low relative to its cardinality, and a uniform distribution among the sites and the levels is achieved, a speed up of nearly n is possible for BOM computations by distribution.

3.3.1 Disadvantage of synchronous control

The synchronous DBOM algorithm can perform well when the actual processing and communication involved in each of the different phases are approximately synchronous, otherwise the synchronous nature of SDBOM can cause long blocking time intervals in the execution at various sites.

The synchrony hoped for above is not guaranteed at all. Actually the inherent asynchrony of a distributed computing environment may often make the synchronous DBOM algorithm specified above to wait for long periods of time at each phase until all the results (i.e. Base) produced by the other sites in that phase to be delivered by the communication subsystem. To avoid such exhaustive waiting while there are much work that could be done, the following asynchronous DBOM algorithm is suggested.

3.4 An asynchronous DBOM algorithm

The asynchronous DBOM (ADBOM) algorithm is similar to the synchronous one except for the part dealing with the receipt of the *Base* fragments. While each site in the synchronous algorithm waits (i.e. delays the start of its next phase) until it receives all the *Base* tuples produced in the current phase by every site, each site in the asynchronous algorithm collects the *Base* tuples that have already arrived and does not wait unless no *Base* tuples has been produced locally in the current phase, and none has yet arrived from the other sites. Notice that such waiting occurs only when there is still some tuples in the local $Uses$, for which the total cost have not yet been computed.

The ADBOM algorithm simulates the same behavior as that of the synchronous one, when the computational phases and the exchange of their results are synchronous. That is, when at all sites and for all phases the result of the previous phase from a site r arrives at a site p before p performs its first *Receive* action.

The intuition behind the well-functioning of the asynchronous DBOM algorithm lies in the fact that the total cost for a part from level i are only dependent on the total cost of some few parts from the levels below. Hence the availability of the total cost for some parts from the levels below may often enable the computing of the total cost for some parts of the level above. This will on the average speed up the overall computation. This effect may be strengthened if the disjunctive groups of tuples located at the different sites are highly independent.

3.5 Implementation of the asynchronous DBOM algorithm

The implementation of the ADBOM algorithm differs from that of the SDBOM in the manner the *Base* fragments are received from other sites.

In the SDBOM implementation the $RecvAll_p^i$ action blocked the execution at site p waiting for the delivery of $Base_r^i$ for each remote site r in the system. In this implementation, we will be using the action $RecvAv_p^i$ which receives all the $Base_r^i$ fragments available locally at the moment this action is called (and before it terminates). A $Base_r^i$ fragment is made available at site p if it has been delivered by the underlying communication subsystem before next phase starts. This is implemented by the statements labeled by 4, 5 and 6 in the algorithm depicted below.

The implementation is based on an asynchronous I/O routine called *Select* that is capable of simulating both blocking and nonblocking I/O behavior. The *Select* operation used here takes two arguments, the first is a list of site identities and the second is a time value determining the waiting period for some data to arrive from some sites. If this time value is zero, no waiting will take place. *Select* returns the site identity from which some data has arrived, or $\perp$ when No_WAIT (i.e. 0) and no data arrived from any site. When a site identity is returned, *Receive* is called to fetch the data that has arrived from that site.

```
**Algorithm** $ADBOM_p$
**Input:** $Accum_p, Uses_p, Base$;
**Begin**
    $i \leftarrow 1$;
    **While**($Uses_p^i \neq \emptyset$) **Do**
        1: $(Accum_p^i, Uses_p^i, Base_p^i) \leftarrow CJOIN(Accum_p^{i-1}, Uses_p^{i-1}, Base^{i-1})$;
        2: $Broadcast_p(Base_p^i)$;
        3: $Base^i \leftarrow Base_p^i$;
        4: **If**($Uses_p^i = \emptyset$) **Then**
               **Goto** Terminate;
        5: **If** $Base_p^i \neq \emptyset$ **Then**
               $aport \leftarrow Select(\mathcal{P}, NO\_WAIT)$;
           **Else**{wait when there is no new base tuples }
               $aport \leftarrow Select(\mathcal{P}, WAIT)$;
        6: **While**($aport \neq \perp$) **Do**
               $(aport, Base_{aport}^i) \leftarrow Receive(\mathcal{P})$;
               $Base^i \leftarrow Base^i \cup Base_{aport}^i$;
               $aport \leftarrow Select(\mathcal{P}, NO\_WAIT)$;
           **Od**; {There is no more base fragments available}
           $i \leftarrow i + 1$;
    **Od**; { while $Uses_p^i$ is not empty }
    Terminate: $Broadcast_p(Terminated)$;
**End**.
```

The ADBOM algorithm specified above breaks with our previous understanding of a phase which was established for the iterative BOM and the synchronous DBOM algorithms, i.e. it breaks the correspondence between a computational phase i (i.e. $Cjoin^i$), and computing the total cost for all i^{th} level parts. However, for the first phase the correspondence still hold, i.e.

$$Cjoin_{p_1}^1; Bcast_{p_1}^1; RecvAv_{p_1}^1; \ldots ; Cjoin_{p_n}^1; Bcast_{p_n}^1; RecvAv_{p_n}^1;$$

The difference between this and the first phase of the SDBOM is that *RecvAv* will try to receive the base tuples sent by remote sites, if they are already delivered locally (i.e. available), otherwise it returns immediately when there is some base tuples produced locally. It will block only when there is no new base tuples produced locally, and no base tuple has yet arrived from some remote sites. This means $Cjoin_{p_j}^2$ could be performed only on a subset of the base tuples (from the first level), and in fact a call of $Cjoin$ may be needed for each of the base fragments sent by remote sites and received and processed locally in a serial manner.

For further analysis see the complexity analysis of ADBOM.

3.5.1 Correctness of the ADBOM algorithm

At first sight, it seemed nearly impossible to prove the correctness of such an asynchronous algorithm. That is due to the tremendous increase in the number of possible executions for the algorithm, caused by its asynchronous nature. Fortunately, it turns out that we can carry an informal proof for the ADBOM algorithm in a manner very similar to that we used for the synchronous algorithm, but with a slight difference.

The first phase computes the total cost for all the parts of the first level, and these parts together with their total cost are broadcast to all the sites in the system. Consequently each site will *eventually* compute the total cost for all its local second level parts, and broadcast them. That is, each site will eventually receive all the second level parts in the entire system together with their total cost. Again, this implies that each site will eventually compute the total cost for all its local third level parts, and broadcast them. We can carry on with this argument until we approach level D parts, which prove that any (fault-free) execution of ADBOM will compute BOM for all the parts in the system.

We need now to prove that the algorithm will terminate after it has computed the total cost for all the parts from level 2 upto D. It follows from the above argument that each site p will compute the total cost for all its local parts and broadcast to all the other sites. This implies that the local $Uses_p$ will be empty after p has computed the total cost for all the local parts from level 2 upto D. Hence, $ADBOM_p$ will terminate properly.

Notice that the eventuality argument from above assumes fairness of the underlying system.

3.6 Complexity Analysis for ADBOM

In a run of ADBOM, the first phase will always consist of the same set of action occurrences, in which each site p performs the sequence of events

$$CJOIN_p^1; Bcast_p^1; RecvAv_p^1;$$

which is partially the same as in the synchronous case.

The ADBOM algorithm will always generate a run fragment similar to the following with the actions of the different sites occurring in any order, and the actions of the same sites are totally ordered as the following fragment shows:

$$Cjoin_{p_1}^1; Bcast_{p_1}^1; RecvAv_{p_1}^1; \ldots ; Cjoin_{p_n}^1; Bcast_{p_n}^1; RecvAv_{p_n}^1;$$

The above fragment corresponds to the execution of the first phase at all the sites in the system.

For comparison, an SDBOM run fragment that corresponds to the first phase looks as follows:

$$Cjoin_{p_1}^1; Bcast_{p_1}^1; \ldots ; Cjoin_{p_n}^1; Bcast_{p_n}^1; RecvAll_{p_1}^1; \ldots ; RecvAll_{p_n}^1;$$

When we compare the two run fragments above, we observe that an $RecvAv_p^1$ action may occur before any $Cjoin_q^1$ action, where $p \neq q$, when $Bcast_p^1 \neq \emptyset$, while a $RecvAll_p^1$ action could never precede a $Bcast_q^1$ (or a $Cjoin_q^1$) action , for any q and p.

In the first phase of ADBOM each site produces all its local first level $Base$ tuples, and disseminates them (by broadcast) to all sites, which is very similar to what happens in the first phase of SDBOM.

In the second phase however the situation is different, because each of the n $Base$ fragments sent in the previous phase (i.e. a $Base_{p_i}^1$ is sent for each $p_i \in \mathcal{P}$) may be received at a site p so asynchronously that for each of them there is a fragment of execution at p consistent of the following:

$$Cjoin_p^{2,p}; Bcast_p^{2,p}; RecvAv_p^2;$$

The ADBOM execution fragment at site p that is computationally equivalent to the second phase of an SDBOM execution, may look as follows:

$$\begin{gathered}
Cjoin_p^{2,p}; Bcast_p^{2,p}; RecvAv_p^{2,p_1};\\
Cjoin_p^{2,p_1}; Bcast_p^{2,p_1}; RecvAv_p^{2,p_2};\\
\vdots\\
Cjoin_p^{2,p_{n-1}}; Bcast_p^{2,p_{n-1}}; RecvAv_p^{2,p_n};\\
Cjoin_p^{2,p_n}; Bcast_p^{2,p_n}; RecvAv_p^{3};
\end{gathered}$$

where an $Cjoin_p^{2,p_i}$ event is enabled by the receipt of $Base_{p_i}^1$, which is the result of the $RecvAv_p^{2,p_i}$ event. We denote the call to CJOIN by $Cjoin_p^{2,p_i}$ to indicate that this join involve only a fragment of the base tuples involved in the previous phase, and this fragment is the one produced and broadcast by site p_i. An $RecvAv_p^{2,p_i}$ event denotes the receipt of the base fragment produced and broadcast by site p_i in the previous phase (i.e. $Base_{p_i}^1$).

The above schedule is very bad because we are paying the price of performing a join for each base fragment and not only one join for the union of all fragments (as in the synchronous case), and the cost of this join is strongly dependent on the size of the local $Uses$ relation. Unfortunately the situation can be much worse by the fact that for each $Base$ fragment (also the one produced locally) a series of D phases could occur before the next $Base$ fragment is received as shown by the following execution fragment at site p:

$$\begin{gathered}
Cjoin_p^1; Bcast_p^1; RecvAv_p^1;\\
Cjoin_p^2; Bcast_p^2; RecvAv_p^2;\\
\vdots\\
Cjoin_p^D; Bcast_p^D; RecvAv_p^D;
\end{gathered}$$

In the above execution fragment, no base fragment is received from a remote site, i.e. all the $RecvAv_p^i$ events returns immediately without receiving anything. Such an execution fragment is enabled by the fact that each of the $Cjoin_p^i$ events produced a local base fragment, and no base fragment was received while the execution was going on.

Such an execution fragment could occur for each base fragment produced in the system irrespective where it has been produced. However, the time period between two subsequent deliveries of (remote) base fragments have to be long enough for such an execution fragment to occur. The following execution fragment simulates the above execution fragment for each of the base fragments produced in the first phase:

$$\begin{gathered}
Cjoin_p^1; Bcast_p^1; RecvAv_p^1;\\
Cjoin_p^2; Bcast_p^2; RecvAv_p^2;\\
\vdots\\
Cjoin_p^D; Bcast_p^D; RecvAv_p^{D,p_1};\\
Cjoin_p^{2,p_1}; Bcast_p^{2,p_1}; RecvAv_p^{2,p_1};\\
\vdots\\
Cjoin_p^{D-1,p_1}; Bcast_p^{D-1,p_1}; RecvAv_p^{D-1,p_1};\\
Cjoin_p^{D,p_1}; Bcast_p^{D,p_1}; RecvAv_p^{D,p_2};\\
\vdots\\
Cjoin_p^{D,p_{n-1}}; Bcast_p^{D,p_{n-1}}; RecvAv_p^{D,p_n};
\end{gathered}$$

$$\begin{gathered} Cjoin_p^{2,p_n}; Bcast_p^{2,p_n}; RecvAv_p^{2,p_n}; \\ \vdots \\ Cjoin_p^{D-1,p_n}; Bcast_p^{D-1,p_n}; RecvAv_p^{D-1,p_n}; \\ Cjoin_p^{D,p_n}; Bcast_p^{D,p_n}; RecvAv_p^{D,p_n}; \end{gathered}$$

In the above execution fragment most of the $RecvAv_p$ events do not succeed in receiving any *Base* fragment. Actually only the following events receive a fragment sent in the first (old) phase:

$$RecvAv_p^{D,p_1}, RecvAv_p^{D,p_2}, \ldots , RecvAv_p^{D,p_n}$$

By analyzing the above fragment, we realize that each of the n *Base* fragments produced and broadcast in the first phase can be engaged in $D-1$ phases of the new sort, in each of which the events *Cjoin*, *Bcast*, and *RecvAv* occur. The *Cjoin* in each of these new phases may produce a new *Base* fragment, which is then broadcast by *Bcast*, and finally an attempt to (asynchronously) receive some *Base* fragments produced and broadcast by other sites, is carried out by the *RecvAv* event.

A maximum total of $n(D-1)$ *Base* fragments can be produced (in the second phase) of which $(n-1)(D-1)$ are received by each site in the third phase (each site need not receive the fragments produced locally and which have already been locally processed completely).

As mentioned earlier, we have broken with our notion of phases as created by the synchronous DBOM algorithm, because parts of all the subsequent phases are executed in the second phase, as shown in the schedule above. What is special about the new perception of a phase is that in ADBOM phase i the cost for all parts of level i are computed completely, and for all the *Base* fragments received in this phase a chain of *Cjoin* and *Bcast* events may occur for each fragment up to the last level $i \leq D$ for which a new nonempty *Base* fragment is created.

The third ADBOM phase consists of the asynchronous receipt of $(n-1)D$ *Base* fragments, each of them triggering an execution fragment similar to the one above but having one less level, because the total cost for all parts from level 2 have been computed completely in the previous (i.e. second) phase. Consequently, the third phase will produce $(n-1)D(D-1)$ *Base* fragments, where each site will receive $(n-1)D(D-1)$ of them.

Below we state for each ADBOM phase the number of *Cjoin* (or *Bcast*) that can take place in an ADBOM execution:

phase	*produced by each site*	*received by each site in next phase*
1	D	$(n-1)D$
2	$(n-1)D(D-1)$	$(n-1)(n-1)D(D-1)$
$\vdots$		
i	$(n-1)^{i-1}D!/(D-i)!$	$((n-1)^i)D!/(D-i)!$
$\vdots$		
D	$(n-1)^{D-1}D!$	$(n-1)^D D!$

The cost formula for such an ADBOM execution at site p (which is the worst case that can occur) is expressed in terms of the number of *Cjoin* events:

$$CF_{ADBOM_p} = \Sigma_{i=1}^{D}(n-1)^{i-1}D!/(D-i)! \tag{7}$$

This is an extremely high cost, since the cost of CJOIN is strongly dependent on the size of *Uses*.

To derive the communication complexity of the worst case execution of ADBOM, recall that an *Bcast* event follows each of the *Cjoin* event generated by an ADBOM execution, thus the maximal number of *Bcast* events is equivalent to that for *Cjoin*. Fortunately however, the communication complexity in terms of the number of base tuples exchanged among the sites are the same as for the SDBOM algorithm, i.e. I base tuples will be exchanged in ADBOM executions irrespective of the number of *Cjoin* or *Bcast* events in an execution. Later on, we will show how to make the number of *Bcast* events in ADBOM executions fixed (independent of the number of *Cjoin* events), and in fact often less than that for SDBOM executions.

3.7 Data distribution and the impact of execution timing

It is relatively trivial to satisfy the condition stated in assumption 3.2, while the condition in assumption 3.3 is very hard to satisfy.

The assumptions 3.2 and 3.3 above are important in a synchronous system, or when approximately synchronous executions (i.e. runs of the algorithm) occur, because they achieve a uniform distribution of the workload at the various sites involved. This is in contrast to asynchronous systems, where there are cases (i.e. possible executions) in which a non-uniform distribution of the data will be preferred. That is, if the sites having more groups of tuples, also have higher processing capacity than those having fewer tuples, a better distribution than the uniform one is achieved [1]. Unfortunately, this can not be guaranteed to occur. The opposite situation could also occur; i.e. sites having higher processing capacity have fewer tuples than those having lower processing capacity, which leads to bad performance, since the slowest processor determines the response time.

When taking load variation into account, the processing capacity of a site is a function of processor speed and system load for that site. The higher speed and lower load a site has, the more processing capacity it has. There is no guarantee that the data distribution will be such that the workload is distributed among the different sites along the different phases. In fact the worst case occurs when the groups of tuples from different levels are located at different sites, resulting in a performance of our algorithm that will be worse than a sequential algorithm, because the resulting computational and communication activities at the various sites will be strictly ordered in time. Moreover, the synchronous algorithm will not function properly for such a distribution.

However, by using a random distribution function, an approximate satisfaction will be achieved with high probability.

Although parallelism is inherent in distributed computing systems, its exploitation is not a straightforward matter. The problem at hand and the design of its solution must allow for parallel execution. However, there is no guarantee that the logically independent activities that are scheduled by the solution to execute in parallel are actually executed in parallel by the underlying distributed system.

In both the synchronous and the asynchronous distributed BOM algorithms the danger of strictly sequential execution of computation and communication activities is always present; i.e. there may never be simultaneous activities at different sites. Although the danger is there, strict sequential execution rarely occurs.

[1] we speak here of the processing capacity throughout a whole run, and not at any specific moment through that run

4 Further DBOM analysis, improvements, and future work

In this section we will discuss a number of improvements to both the SDBOM and the ADBOM algorithms. These improvements will be presented and discussed one by one. However, they can be combined into a single (improved) ADBOM or SDBOM algorithm.

From equation 7, we see that a very large number of $Cjoin$ events could occur in an ADBOM execution. However, in ADBOM they are carried out when an SDBOM execution will be blocked while waiting for all the base tuples produced in the previous (SDBOM) phase to be available locally.

Each of these $Cjoin$ events has the potential of reducing $Uses$ and producing new base tuples that are broadcast by a subsequent $Bcast$ event, hence making them available for sites having parts with cost dependent on these base tuples. This *early availability* of the base tuples made possible by the ADBOM algorithm can improve the performance and overall throughput of these sites. The ADBOM algorithm enables a better progress in the computation. Some sites may be able to compute the total costs for all their local parts based on the receipt of some (recently computed) base parts from other sites, while SDBOM may be blocking such progress. This is based on the assumption that the sub-DAGs at the different sites does not involve all the parts in the system (in a subpart role).

The ADBOM algorithm avoids blocking and enable the early delivery of base tuples at the cost of increased number of $Cjoin$ events, which will potentially improve the response time of the algorithm at the cost of increasing its total cost. However, since the cost of a $Cjoin$ event is strongly dependent on the current cardinality of $Uses$, this large number of $Cjoin$ events will most probably decrease the overall system performance and throughput to an unacceptable level.

4.1 Minimizing the cost of CJOIN

Our first step to improve the performance of ADBOM is to redesign CJOIN to make it strongly dependent on M, the cardinality of current $Base$, and weaken its dependency on N, the cardinality of current $Uses$.

When the available base tuples (i.e. the tuples that were in the recently delivered $Base$ fragments) to be engaged in the next $Cjoin$ event are very few compared to the number of tuples currently in $Uses$ (which is very typical for ADBOM), ADBOM will use another combined join algorithm than CJOIN, when carrying out an $Cjoin$ event. This other algorithm (from now on called DJOIN) differs from CJOIN in that it is strongly dependent on M and not as strongly dependent on N as CJOIN. Since M is much less than N, a call to DJOIN costs much less than its corresponding call to CJOIN, hence improving the efficiency (i.e. response time and total cost) of ADBOM.

This DJOIN algorithm requires an index to be built on $Uses$, based on the *subpart* attribute, in addition to the structures required by CJOIN on $Uses$, $Base$, and $Accum$. The DJOIN algorithm is very similar to CJOIN except that instead of engaging all the tuples of $Uses$ it engages only those $Uses$ tuples that have as its *subpart* one of the currently engaged $Base$ tuples.

Notice that a base tuple could be engaged, as a subpart, in many tuples (hence data blocks) of $Uses$, and many base tuples could be subparts of the same part, hence retrieving the same data block of $Uses$ from disk could satisfy many base tuples. This implies that a large $Base$ (engaged in a join event) results in fewer disk accesses needed in performing the join event. Therefore we should minimize the number of join events to enlarge the $Base$ relation involved in them.

4.2 Minimizing the number of joins: a dynamic approach

To minimize the number of join events involved in a run of ADBOM, we restrict the degree of nondeterminism (i.e. asynchrony of computational phases) found in such a run (or weaken the phase synchrony of SDBOM, i.e. increasing its nondeterminism). This restriction is materialized by some amount of waiting. That is, the algorithm waits for the results (i.e. base fragments) of at least K sites to be delivered locally before going on with the next join event (i.e. phase). However, this waiting must be restricted by $1/c$ times the amount of time it takes a join event to be carried out, for a suitable $c > 1$.

Both K and c can be dynamically adjusted by the current system load as follows: When there is high load (i.e. many other activities are going on simultaneously with our BOM computation) it is beneficial to wait (i.e. make K large and c small) otherwise do not wait much. Such a dynamic synchrony-adjusting mechanism depending on system load has the potential of increasing system throughput. It remains to be shown what effect such a mechanism will have on the response-time of our BOM computation.

In future work, intensive experimental activities have to be carried out to measure the "real" impact of K, c and the dynamic synchrony-adjusting mechanism on a number of data and topological settings.

4.3 Restricting the size of $Base$ to available memory

Since the cost of each $Cjoin$ event is dependent on the current cardinality of $Uses$, i.e. $|Uses| = N$, and not the current cardinality of $Base$, i.e. $|Base| = M$ [2], it is beneficial to make the $Base$ relation that is engaged in a $Cjoin$ event as large as possible (even by some waiting). This will increase the number of base tuples produced by $Cjoin$, and increase the reduction rate for $Uses$, which again make subsequent $Cjoin$ events cheaper. However, if the size of $Base$ gets larger than available memory, it is more efficient to execute many $Cjoin$ events than having to apply even the most efficient non main memory hash join algorithm, i.e. hybrid hashing [Ullm88b, Graf93]. The cost of hybrid hash join which is applied when non of the argument relations fits into memory is $3(M + N)$, while the cost of a main memory hashing algorithm (which is applied when at least one of the argument relations fits into available memory) is $max(N, M)$.

That is, in addition to the current system load the amount of available memory has a major impact on the actual figures of K and c.

4.4 When broadcast is not necessary

Using broadcast to disseminate messages of base tuples to all sites in the broadcast domain, is beneficial when all the sites are interested in these messages. It is often the case that some sites are not interested in all the messages, because their base tuples are not engaged with any local part. Since these sites cannot know beforehand when the base tuples of a message are needed or not, they have to receive them and handle them in the same manner as if they were interested in them.

When the degree of independence between the subsets of groups located at the various sites is very high (i.e. the base tuples produced locally are seldom needed by other sites), it is a great waste of resources to use broadcast, since a broadcast will unnecessarily engage all the sites, even those having nothing to do with our BOM computation.

[2] provided that it fits into main memory

Fortunately, in such cases it is possible to avoid broadcast and use simple point-to-point communication, but there is a need to know which sites are dependent on the cost of which remote parts (i.e. parts located at a remote site). The solution to this problem is simple, since the partition scheme is a priori known to each site, then the owner sites of parts are globally known. Therefore, at the start of each run, each site sends to sites (on which parts it is dependent) the set of parts identities it is interested in getting their total cost. And, further in the run, when a site produces the base tuples of its current phase it only sends a subset of the base parts to the sites that have submitted request for them. When the subsets of groups at the various sites are totally independent, no site ever sends any base tuple to any other site.

The above idea is also applicable for synchronous BOM algorithms, and for point-to-point networks.

4.5 Pipelining

Using the pipelining strategy in distributed data-processing, a site disseminates the base tuples recently produced locally at that site as soon as they are produced. This strategy is space efficient, disperses the transfer of bulk data over a large period, and increases parallelism in the system, hence increasing the throughput of the communication subsystems. Most importantly, it increases the amount of parallel activities among the sites and the network channels, hence improving the response time of the computation and the overall throughput of the processing system. For a comprehensive analysis of pipelining in distributed data processing systems see [Khal96].

5 Conclusions

Linearly recursive database queries can be expressed as a combination of a transitive closure and relational algebraic operations. In this paper we have presented efficient and distributed algorithms for an important class of such queries, called bill-of-materials (BOMs). A BOM query can be expressed as a combination of transitive closure and aggregate functions.

The presented algorithms use a special join operation that avoids the evaluation of the transitive closure but rather combines the accumulation of the closure with the binary matching of the join into a more efficient algorithm.

For the distributed setting we presented two algorithms, a synchronous and an asynchronous one, and suggested some improvements to both of them. The synchronous algorithm consists of a small number of phases (i.e. join events) but at the potential cost of long waiting at the end of each phase. While the asynchronous algorithm eliminates waiting (while there is some work to be done) at the potential cost of a greater number of phases. The worst case scenario for the asynchronous algorithm can involve a very large number of join events, which may decrease system performance tremendously.

As a consequence of the above analysis, it seems to us that a partially synchronous approach that is adaptive to the actual execution environment may be the best solution to avoid the two stated problems; i.e. long waitng caused by synchronous phases, and the large number of join events caused by asynchrony. Moreover, a redesign of the join algorithm was proposed to minimize the dependence of the execution cost on the number of join events. That is in addition to some other proposed improvements.

The experimental verification of these claims is the topic of our immediate future work.

References

[Agra87] Agrawal, R., "Alpha: An extension of Relational Algebra to Express a class of Recursive Queries," Proc. 3rd Int'l Conf. on Data Engineering, February 1987.

[AgJa87] Agrawal, R., Jagadish, H.V.,"Direct Algorithms for Computing the Transitive Closure of database relations," in Proc. 13th Int'l Conf. on VLDB, 1987.

[AgJa88] Agrawal, R., Jagadish, H.V.,"Multiprocessor Transitive Closure Algorithms," in Proc. Int'l Symp. on Databases in Parallel and Distributed Systems, Austin, Texas, Dec. 1988, pp. 56-67.

[AgDJ90] Agrawal, R., Dar, S., Jagadish, H.V., "Direct Transitive Closure Algorithms: Design and Performance Evaluation," in ACM Trans. on Database Systems, 15(3), Sept. 1990.

[Banc85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," TR. DB-004-85, MCC, Austin, Texas, 1985.

[BiSt88] Biskup, Stiefeling, " Transitive Closure Algorithms for Very Large Databases," TR, Hochschule Hildesheim,1988.

[Brat84] Bratbergsengen, K., "Hashing Methods and Relational Algebra Operations," Int'l Conf. on VLDB, Singapore, Aug. 1984.

[ChDe90] Cheiney, J., De Maindreville, C., " A Parallel Strategy for the Transitive Closure Using Double Hash-based Clustering," in Proc. Int'l Conf. on VLDB, aug., 1990.

[ChHa82] Chandra, A.K., Harel, D., "Horn clauses and the fix-point query hierarchy," Proc. 1st Symp. Principles of Database Systems, 1982, pp. 158-163.

[Codd70] Codd, E.F., "A relational model of data for large shared data banks," CACM, vol. 13, June 1970, pp. 377-387.

[Codd72] Codd, E.F., "Relational completeness of database sub-languages," DataBase Systems, R. Rustin, Ed. Englewood Cliffs, NJ:Prentice-Hall, 1972, pp. 65-98.

[DaJa92] Dar, S., Jagadish, H.V., "A Spanning tree Transitive Closure Algorithm," in Proc. 8th Int'l IEEE Conf. on Data Engineering, 1992.

[Fran86] Francez, N., Fairness, Springer-Verlag, Berlin, 1986.

[Graf93] Graefe, G., "Query Evaluation Techniques for Large Databases" ACM Computing Surveys 25 (2), June 1993.

[HoAC90] Houtsma, M., Apers, P., Ceri, S., " Distributed Transitive Closure Computation: the Disconnection Set Approach," in Proc. 16th Int'l Conf. on VLDB, Aug., 1990.

[Ioan86] Ioannidis, Y.E., "On the Computation of the Transitive Closure of Relational Operators," Proc. 12th Int'l. Conf. on Very Large Date Bases, August 1986, pp. 403-411.

[JaAN87] Jagadish, H. V., Agrawal, R., Ness, L., "A Study of Transitive Closure as a Recursion Mechanism," Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data, May 1987.

[IaRa88] Ioannidis, Y.E., Ramakrishnan, R., "Efficient Transitive Closure Algorithms," in Proc. 14th Int'l Conf. on VLDB, 1988.

[JaKo84] Jarke, M., Koch, J., "Query optimization in database systems," ACM Computing Surveys, 16(2), pp 111-152, June 1984.

[Gutt84] Guttman, A., "New Features for Relational Database Systems to Support CAD Applications," Computer Science Dept., Univ. of California, Berkeley, June 1984, Ph.D. Dissertation.

[Jako91] Jakobsson, H., "Mixed-approach Algorithms for Transitive Closure," in Proc. of ACM Symp. on PODS, Denver, Co., May, 1991.

[Jian90] Jiang, B., " A Suitable Algorithm for Computing Partial Transitive Closures in Databases," in Proc. IEEE Conf., on Data Engineering, 1990.

[Khal96] Khalaila, A., "Partial Evaluation and Early Delivery: Adapting the Pipeline Processing Strategy to Asynchronous Networks," PhD Dissertation, The University of Tromsoe, 1996.

[Kits83] Kitsuregawa, M., et al., "Applications of Hash to Data Base Machine and Its Architecture," in New Generation Computing, vol. 1, 1983.

[Lu87] Lu, H., " New Strategies for Computing the Transitive Closure of Database Relations," in Proc. 13th Int'l Conf. on VLDB, 1987.

[SACLP79] Selinger, P. G., Astrahan, M.M., Chamberlin, D.D., Lorie, R. A., Price, T.G., "Access Path Selection in a Relational Database Management System," ACM-SIGMOD 1979.

[SmCh75] Smith, J.M., Chang, P.Y., "Optimizing the Performance of a Relational Algebra Database Interface" CACM 18(10), October 1975.

[Tarj81] Tarjan, "Fast Algorithms for Solving Path Problems," Journal of the ACM, 28(3), 1981.

[Ullm88a] Ullman, J.D., Principles of Database and Knowledge Base Systems, Vol. I, Computer Science Press, Rockville, Md., 1988.

[Ullm88b] Ullman, J.D., Principles of Database and Knowledge Base Systems, Vol. II, Computer Science Press, Rockville, Md., 1988.

[VaKh88a] Valduriez, P., Khoshafian, S., "Transitive Closure of Transitively Closed Relations," 2nd Int'l Conf. on Expert Database Systems, L. Kerschberg (ed.), Menlo Park, Calif., Benjamin-Cummings, 1988, pp. 377-400.

[VaKh88b] Valduriez, P., Khoshafian, S., "Parallel Evaluation of the Transitive Closure of a Database Relation," Int'l Journal of Parallel Programming, Vol. 17, No. 1, Feb., 1988.

[Zloo75] Zloof, M.M., "Query-By-Example: Operations on the Transitive Closure," RC 5526, IBM, Yorktown Hts, New York, 1975.